In [1]:
```
print("VELLA PULSER DATA ANALYSIS")
```

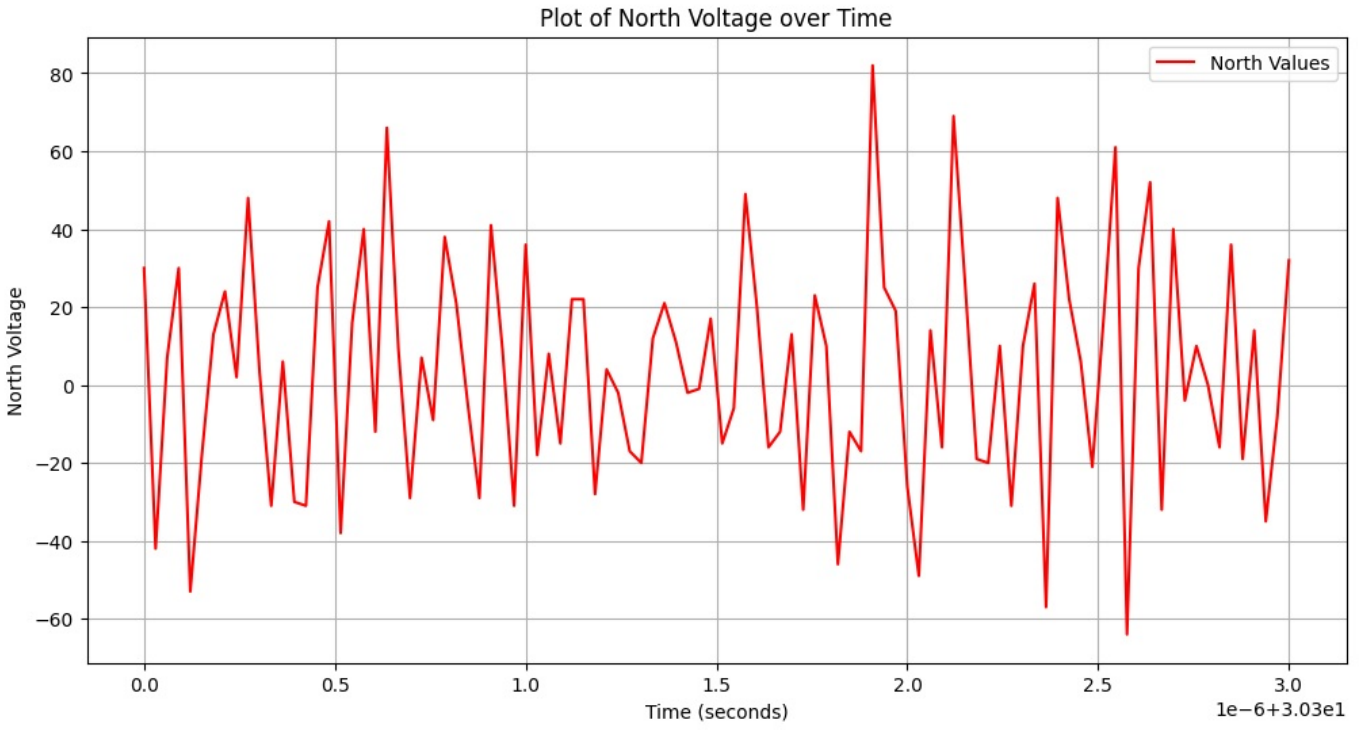
```
VELLA PULSER DATA ANALYSIS
```

In [2]:
```
#STEP1=IMPORTING NECESSARY PACAKAGES

import os
import math
import numpy as np
import pandas as pd
import scipy
import matplotlib.pyplot as plt
from scipy.stats import norm
from scipy.stats import expon
from scipy.stats import chi2
import seaborn as sns
from numpy.fft import fft
from scipy.optimize import minimize
from scipy.signal import correlate
from scipy.ndimage import gaussian_filter
from scipy.signal import find_peaks, peak_prominences
from scipy.ndimage import shift
```

In [3]:
```
#STEP=2 =READING THE DATA
df=pd.read_csv("jin.txt", delimiter=" ",header=None ,names=["N","S"])
north = df.iloc[:100,0].values
south=df.iloc[:100,0].values
```

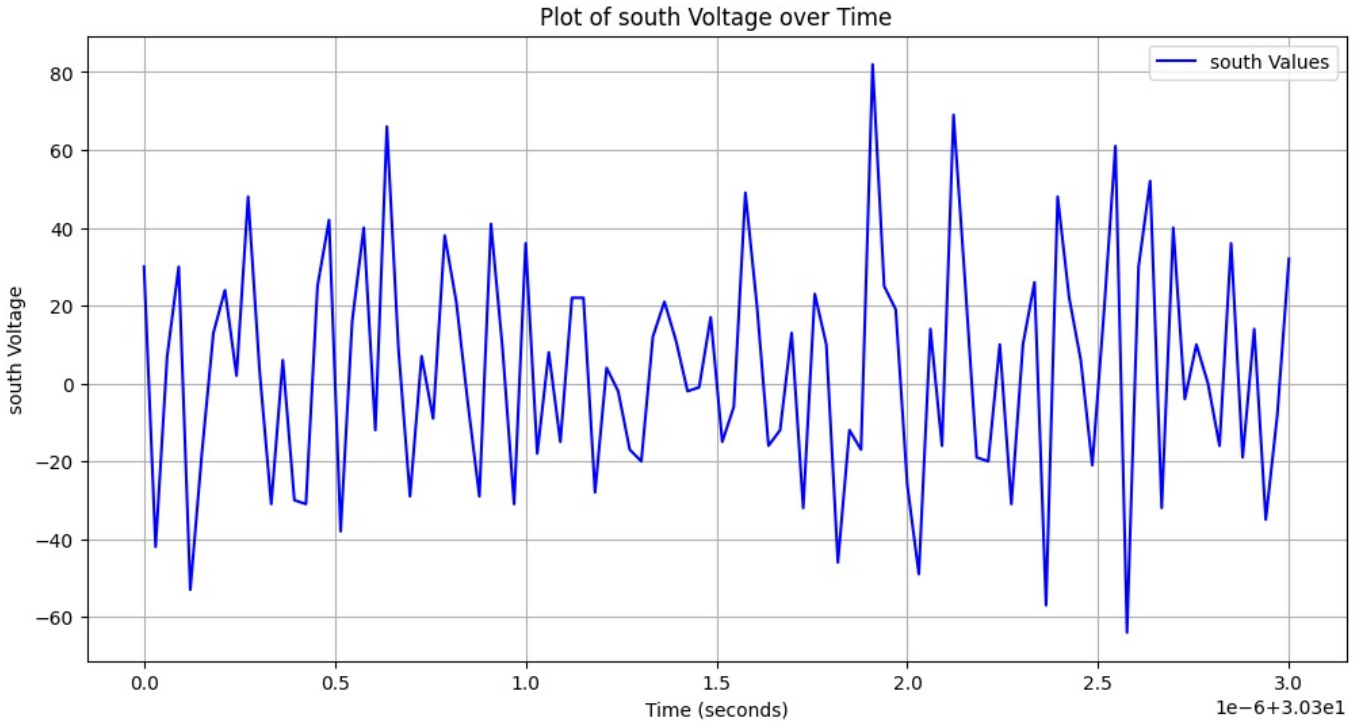
In [4]:
```
#STEP3=TIME AXIS
# Number of rows in the DataFrame
num_rows = len(north)
num_pows = len(south)
# Create time intervals starting from 30.30 seconds
start_time = 30.30  # Start time in seconds
time_intervals = start_time + np.arange(num_rows) * 30.30e-9  # 30.30 nanoseconds interval
secondtime_intervals = start_time + np.arange(num_pows) * 30.30e-9
```

In [5]:
```
# STEP4=PLOT THE DATA OF NORTH ARRAY WITH TIME
plt.figure(figsize=(12, 6))
plt.plot(time_intervals, north, linestyle='-', color='r', label='North Values')
plt.xlabel('Time (seconds)')
plt.ylabel('North Voltage')
plt.title('Plot of North Voltage over Time')
plt.grid(True)
plt.legend()
plt.show()
```



In [6]:
```
# STEP5=PLOT THE DATA OF SOUTH ARRAY WITH TIME
plt.figure(figsize=(12, 6))
plt.plot(secondtime_intervals, south, linestyle='-', color='b', label='south Values')
plt.xlabel('Time (seconds)')
plt.ylabel('south Voltage')
```

```
plt.title('Plot of south Voltage over Time')
plt.grid(True)
plt.legend()
plt.show()
```

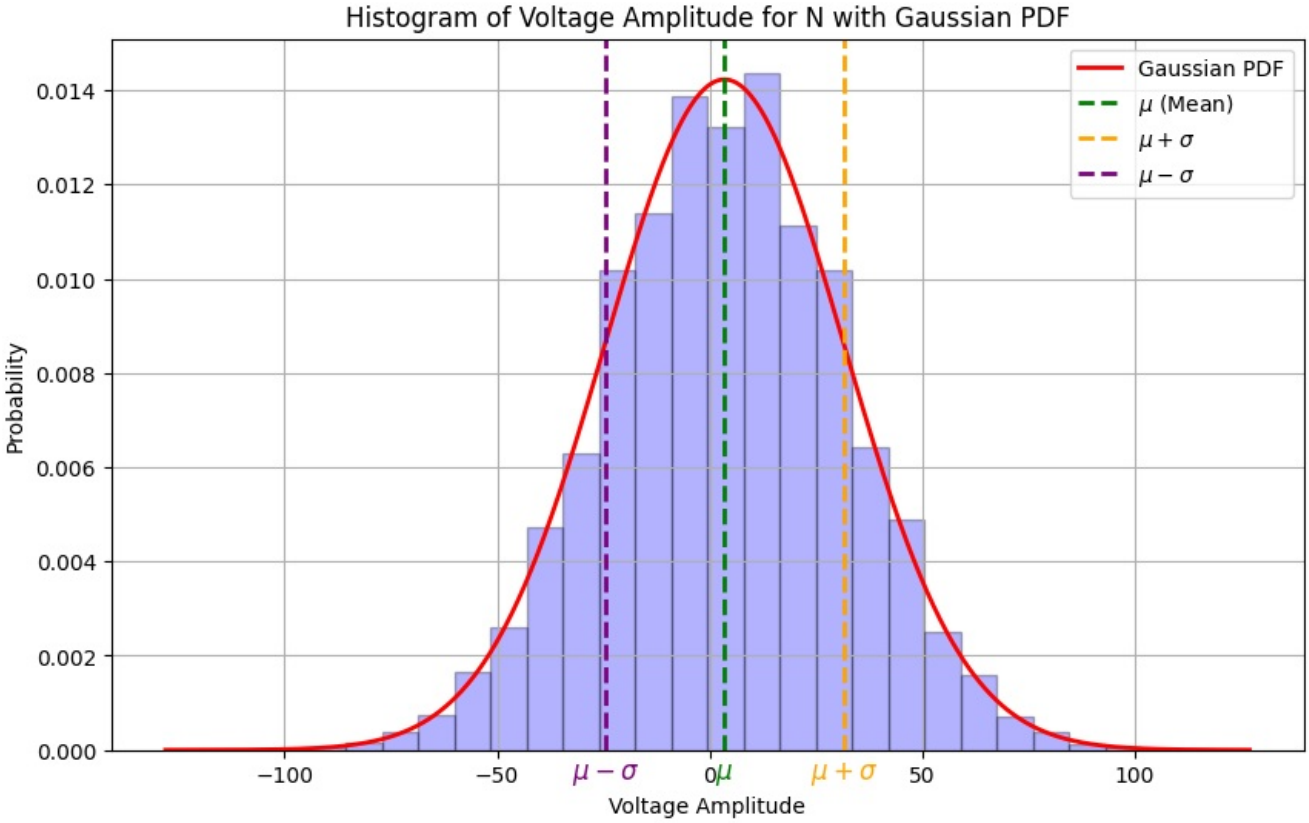


In [7]:
```
#STEP6=CALCULATING STATISTICAL PARAMETERS OF NORTH AND SOUTH DATA


# Extract data
n_data = df["N"].values
s_data = df["S"].values

# Number of data points
N_n = len(n_data)
N_s = len(s_data)

# Mean calculation
mean_n = np.sum(n_data) / N_n
mean_s = np.sum(s_data) / N_s

# Variance calculation
variance_n = np.sum((n_data - mean_n) ** 2) / N_n
variance_s = np.sum((s_data - mean_s) ** 2) / N_s

# Standard Deviation calculation
std_dev_n = np.sqrt(variance_n)
std_dev_s = np.sqrt(variance_s)

# Print the results
print(f"Statistics for column 'N':")
print(f"Mean: {mean_n}")
print(f"Variance: {variance_n}")
print(f"Standard Deviation: {std_dev_n}")

print(f"\nStatistics for column 'S':")
print(f"Mean: {mean_s}")
print(f"Variance: {variance_s}")
print(f"Standard Deviation: {std_dev_s}")
```

```
Statistics for column 'N':
Mean: 3.4861547526041665
Variance: 786.2793720461041
Standard Deviation: 28.040673530536033

Statistics for column 'S':
Mean: 0.73469189453125
Variance: 891.0315163292267
Standard Deviation: 29.85015102690817
```

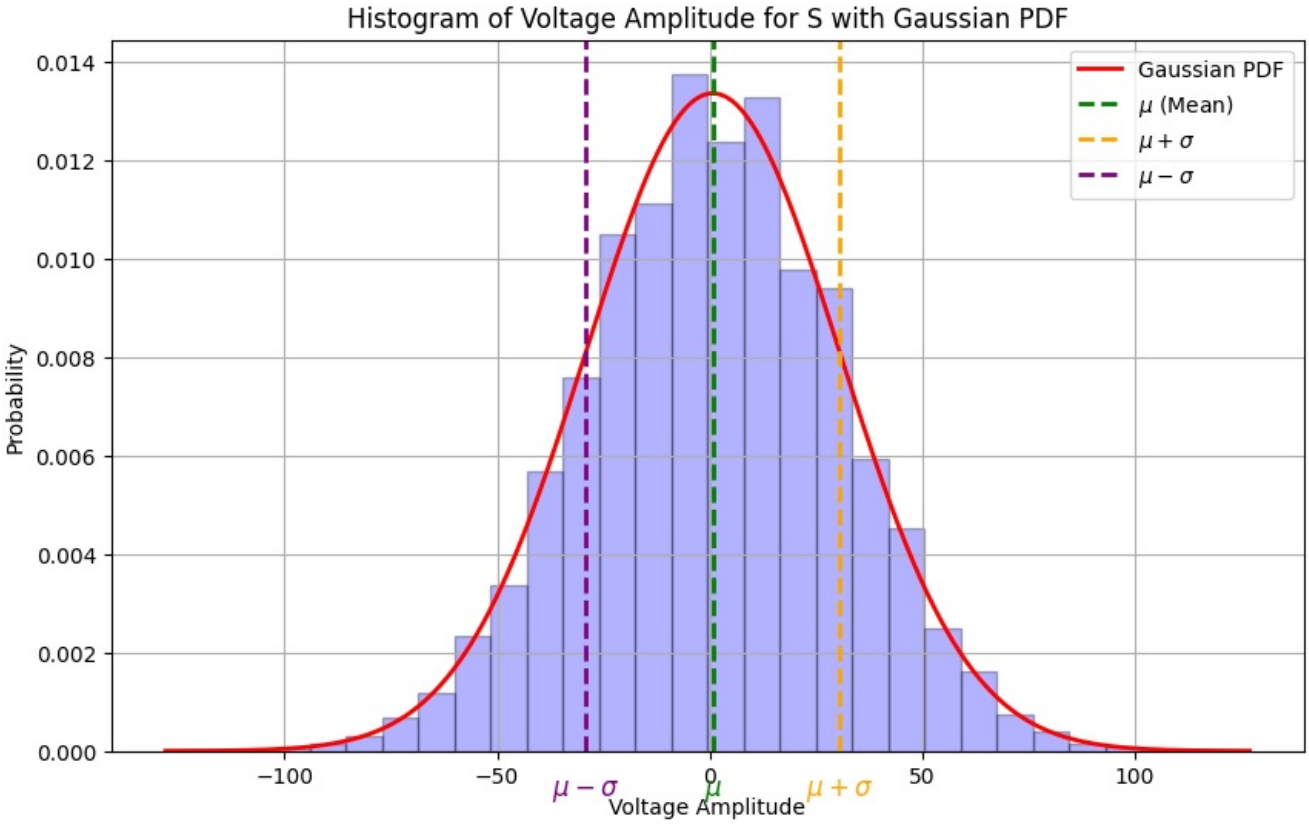
In [8]:
```
#STEP7 =PLOTTING HISTOGRAM AND PDF FOR NORTH DATA


# Statistics for column 'N'
mean_n = 3.4861547526041665
std_dev_n = 28.040673530536033
```

```
# Plot histogram for 'N'
plt.figure(figsize=(10, 6))
n_bins = 30  # Number of bins

# Plot histogram
count, bins, ignored = plt.hist(n_data, bins=n_bins, density=True, color='blue', edgecolor='black', alpha=0.3)

# Plot Gaussian PDF
x = np.linspace(min(n_data), max(n_data), 1000)
pdf = norm.pdf(x, mean_n, std_dev_n)
plt.plot(x, pdf, 'r', linewidth=2, label='Gaussian PDF')

# Mark mean and standard deviations
plt.axvline(mean_n, color='green', linestyle='--', linewidth=2, label=r'$\mu$ (Mean)')
plt.axvline(mean_n + std_dev_n, color='orange', linestyle='--', linewidth=2, label=r'$\mu + \sigma$')
plt.axvline(mean_n - std_dev_n, color='purple', linestyle='--', linewidth=2, label=r'$\mu - \sigma$')

# Annotate mean and standard deviations outside the x-axis

offset = -0.0005 # Adjust this value to position the text outside the graph

plt.text(mean_n, offset, r'$\mu$', color='green', fontsize=12, ha='center', va='center')
plt.text(mean_n + std_dev_n, offset, r'$\mu + \sigma$', color='orange', fontsize=12, ha='center', va='center')
plt.text(mean_n - std_dev_n, offset, r'$\mu - \sigma$', color='purple', fontsize=12, ha='center', va='center')


# Add labels and title
plt.xlabel('Voltage Amplitude')
plt.ylabel('Probability')
plt.title('Histogram of Voltage Amplitude for N with Gaussian PDF')
plt.grid(True)
plt.legend()
plt.show()
```



In [9]:
```
#STEP8 =PLOTTING HISTOGRAM AND PDF FOR SOUTH DATA


# Statistics for column 'S'
mean_s = 0.73469189453125
std_dev_s = 29.85015102690817

# Plot histogram for 'S'
plt.figure(figsize=(10, 6))
s_bins = 30  # Number of bins

# Plot histogram
count, bins, ignored = plt.hist(s_data, bins=s_bins, density=True, color='blue', edgecolor='black', alpha=0.3)

# Plot Gaussian PDF
```

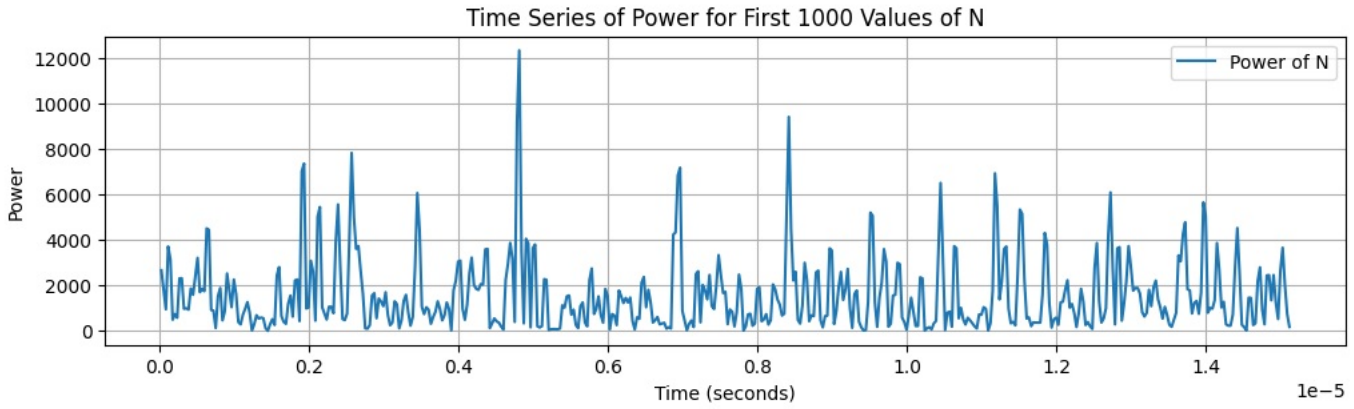
```
x = np.linspace(min(s_data), max(s_data), 1000)
pdf = norm.pdf(x, mean_s, std_dev_s)
plt.plot(x, pdf, 'r', linewidth=2, label='Gaussian PDF')

# Mark mean and standard deviations
plt.axvline(mean_s, color='green', linestyle='--', linewidth=2, label=r'$\mu$ (Mean)')
plt.axvline(mean_s + std_dev_s, color='orange', linestyle='--', linewidth=2, label=r'$\mu + \sigma$')
plt.axvline(mean_s - std_dev_s, color='purple', linestyle='--', linewidth=2, label=r'$\mu - \sigma$')

# Annotate mean and standard deviations outside the x-axis

offset = -0.0008 # Adjust this value to position the text outside the graph

plt.text(mean_s, offset, r'$\mu$', color='green', fontsize=12, ha='center', va='center')
plt.text(mean_s + std_dev_s, offset, r'$\mu + \sigma$', color='orange', fontsize=12, ha='center', va='center')
plt.text(mean_s - std_dev_s, offset, r'$\mu - \sigma$', color='purple', fontsize=12, ha='center', va='center')


# Add labels and title
plt.xlabel('Voltage Amplitude')
plt.ylabel('Probability')
plt.title('Histogram of Voltage Amplitude for S with Gaussian PDF')
plt.grid(True)
plt.legend()
plt.show()
```



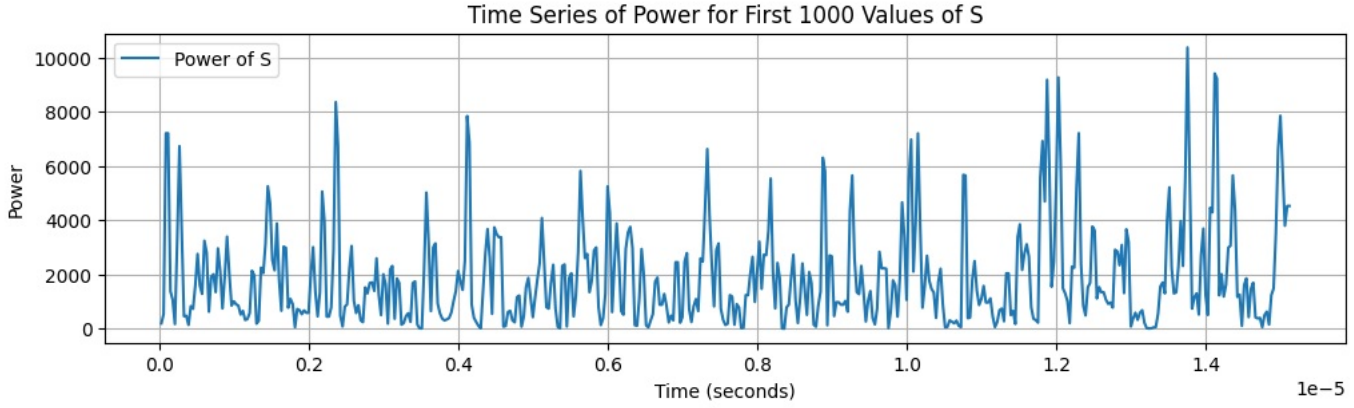
In [10]:

```
#STEP 9=PLOTTING TIME SERIES FOR  POWER OFNORTH DATA



# Extract the first 100 values from column 'N'
n_data = df["N"].iloc[:500].values

# Compute power: square each value and sum successive pairs
n_power = np.array([n_data[i]**2 + n_data[i+1]**2 for i in range(len(n_data) - 1)])

# Time intervals (30.30 ns) and starting at 30.30 ns
time_interval = 30.30e-9  # 30.30 nanoseconds in seconds
start_time = 30.30e-9  # Start at 30.30 nanoseconds in seconds

# Generate time series
time_series = np.arange(start_time, start_time + len(n_power) * time_interval, time_interval)

# Plot time series for 'N' power
plt.figure(figsize=(12, 3))
plt.plot(time_series, n_power, label='Power of N')
plt.xlabel('Time (seconds)')
plt.ylabel('Power')
plt.title('Time Series of Power for First 1000 Values of N')
plt.grid(True)
```

```
plt.legend()
plt.show()
```



In [11]:
```
#STEP10 =PLOTTING TIME SERIES FOR  POWER OF SOUTH DATA



# Extract the first 1000 values from column 'S'
s_data = df["S"].iloc[:500].values

# Compute power: square each value and sum successive pairs
s_power = np.array([s_data[i]**2 + s_data[i+1]**2 for i in range(len(s_data) - 1)])

# Time intervals (30.30 ns) and starting at 30.30 ns
time_interval = 30.30e-9  # 30.30 nanoseconds in seconds
start_time = 30.30e-9  # Start at 30.30 nanoseconds in seconds

# Generate time series
time_series = np.arange(start_time, start_time + len(s_power) * time_interval, time_interval)

# Plot time series for 'N' power
plt.figure(figsize=(12, 3))
plt.plot(time_series, s_power, label='Power of S')
plt.xlabel('Time (seconds)')
plt.ylabel('Power')
plt.title('Time Series of Power for First 1000 Values of S')
plt.grid(True)
plt.legend()
plt.show()
```

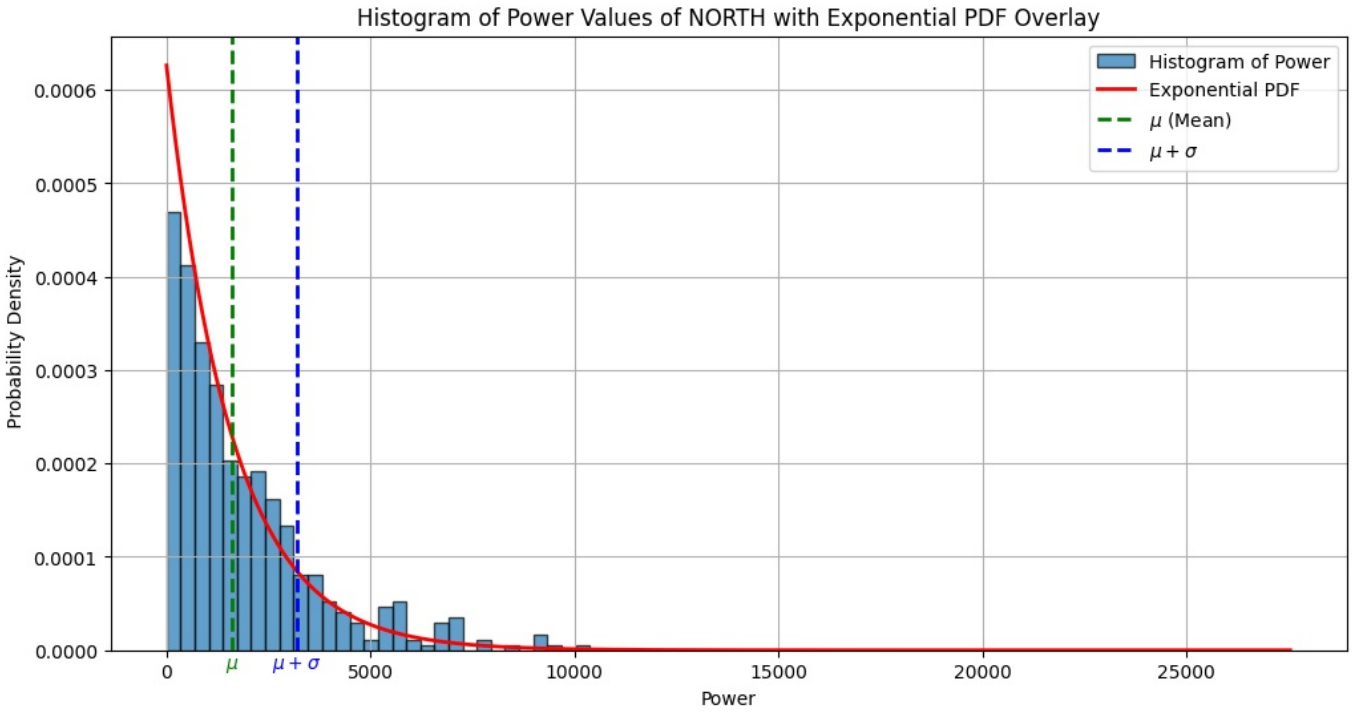


In [12]:
```
#STEP11 =CALCULATING STATISTICAL PARAMEYTERS  POWER OFNORTH DATA

# Extract data from column 'N'
voltage_n = df["N"].values

# Compute power: square each value and sum successive pairs
n_power = np.array([voltage_n[i]**2 + voltage_n[i+1]**2 for i in range(len(voltage_n) - 1)])

# Calculate mean using the formula
mean_n_power = np.sum(n_power) / len(n_power)

# Calculate variance using the formula
variance_n_power = np.sum((n_power - mean_n_power)**2) / len(n_power)

# Calculate standard deviation using the formula
std_dev_n_power = np.sqrt(variance_n_power)

# Print the results
print("statistics of N power")
print(f"Mean of Power: {mean_n_power}")
```

```
print(f"Variance of Power: {variance_n_power}")
print(f"Standard Deviation of Power: {std_dev_n_power}")
```

```
statistics of N power
Mean of Power: 1596.8653072872821
Variance of Power: 2552421.2579250284
Standard Deviation of Power: 1597.6298876539047
```

In [13]:
```
#STEP12 =PLOTTING HISTOGRAM AND PDF OF  POWER OFNORTH DATA


# Statistics for the power values
mean_n_power = 1596.8653072872821
std_dev_n_power = 1597.6298876539047

# Plot histogram of power values
plt.figure(figsize=(12, 6))
plt.hist(s_power, bins=30, density=True, edgecolor='black', alpha=0.7, label='Histogram of Power')

# Plot Exponential PDF
x = np.linspace(min(n_power), max(n_power), 1000)
pdf = expon.pdf(x, scale=mean_n_power)
plt.plot(x, pdf, 'r', linewidth=2, label='Exponential PDF')

# Mark mean and +1 standard deviation on the plot
plt.axvline(mean_n_power, color='g', linestyle='--', linewidth=2, label=r'$\mu$ (Mean)')
plt.axvline(mean_n_power + std_dev_n_power, color='b', linestyle='--', linewidth=2, label=r'$\mu + \sigma$')

# Annotate mean and standard deviations outside the x-axis

offset = -0.00002 # Adjust this value to position the text outside the graph
plt.text(mean_n_power, offset, r'$\mu$', color='g', horizontalalignment='center')
plt.text(mean_n_power + std_dev_n_power, offset, r'$\mu + \sigma$', color='b', horizontalalignment='center')

# Labels and title
plt.xlabel('Power')
plt.ylabel(' Probability Density')
plt.title('Histogram of Power Values of NORTH with Exponential PDF Overlay')
plt.legend()
plt.grid(True)

# Show plot
plt.show()
```



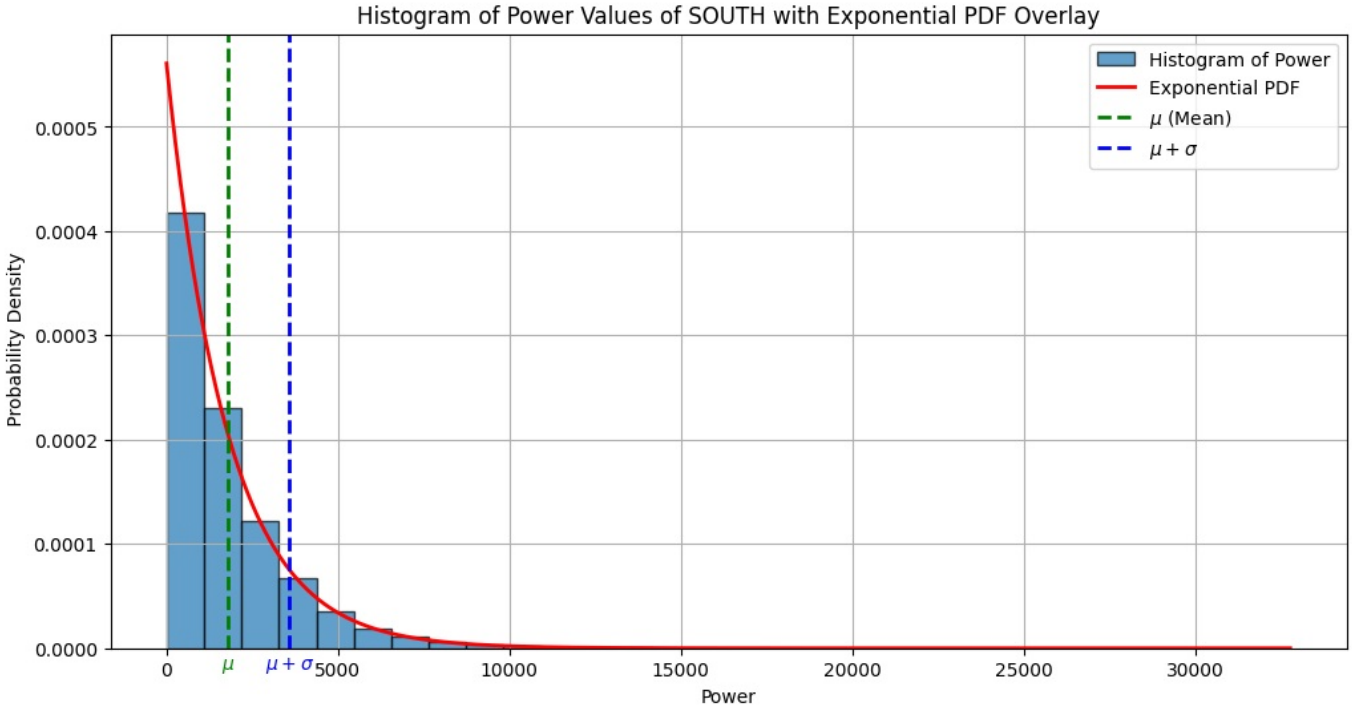
In [14]:
```
#STEP13 =CALCULATING STATISTICAL PARAMEYTERS  POWER OF SOUTH DATA
# Extract data from column 'S'
voltage_s = df["S"].values

# Compute power: square each value and sum successive pairs
s_power = np.array([voltage_s[i]**2 + voltage_s[i+1]**2 for i in range(len(voltage_s) - 1)])

# Number of power values
N = len(s_power)

# Compute the mean of the power values using the formula
```

```
mean_s_power = np.sum(s_power) / N

# Compute the variance of the power values using the formula
variance_s_power = np.sum((s_power - mean_s_power)**2) / N

# Compute the standard deviation
std_dev_s_power = np.sqrt(variance_s_power)

# Print the results
print(f"Statistics for column 'S':")
print(f"Mean of Power: {mean_s_power}")
print(f"Variance of Power: {variance_s_power}")
print(f"Standard Deviation of Power: {std_dev_s_power}")
```

```
Statistics for column 'S':
Mean of Power: 1783.1425970424023
Variance of Power: 3245366.8910109308
Standard Deviation of Power: 1801.4901862099973
```

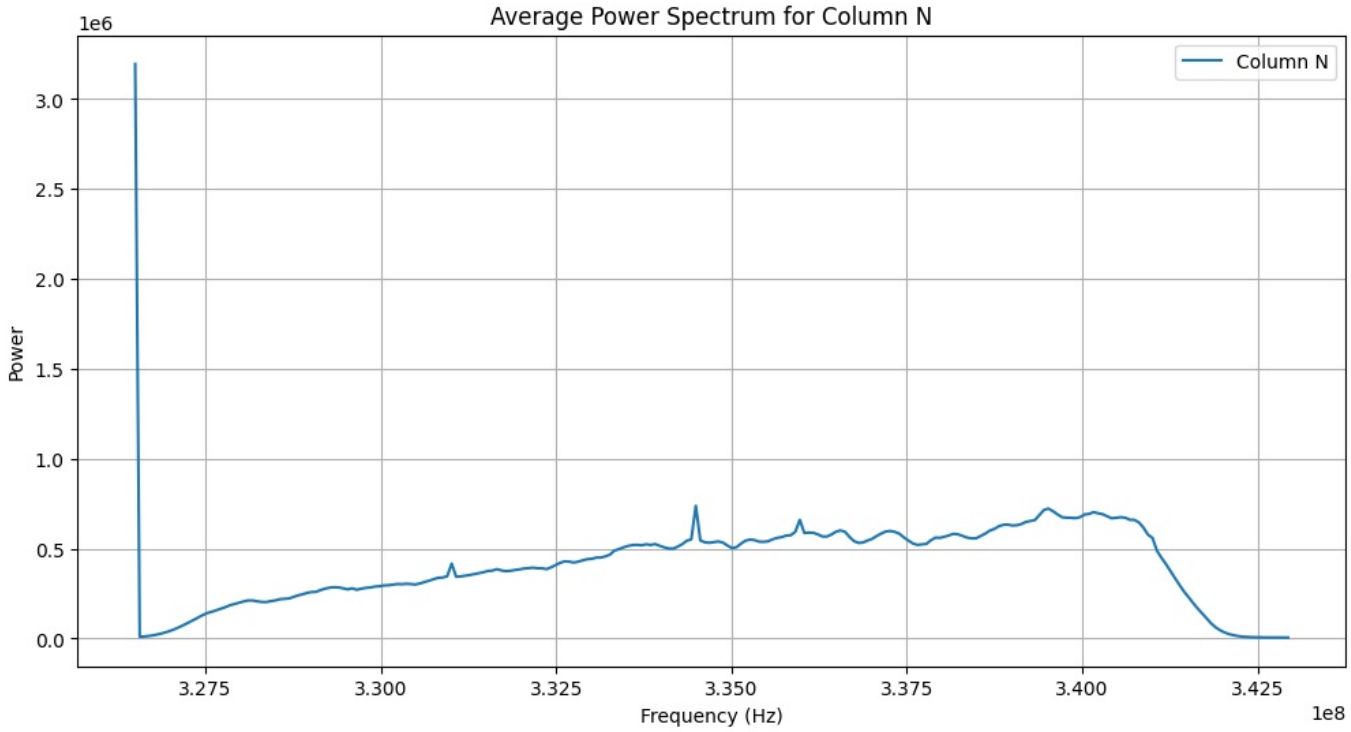
In [15]:

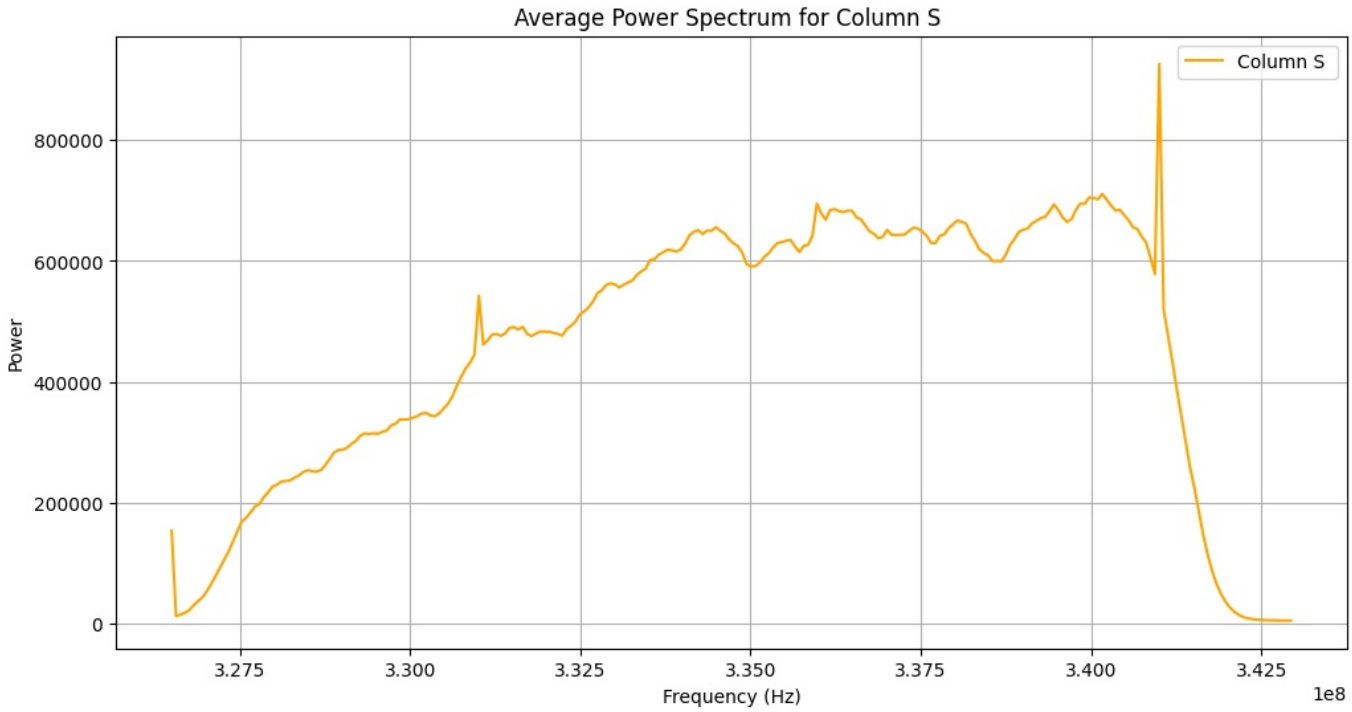
```
#STEP14 =PLOTTING HISTOGRAM AND PDF OF  POWER OF SOUTH DATA



# Statistics for the power values
mean_s_power = 1783.1425970424023
std_dev_s_power = 1801.4901862099973

# Plot histogram of power values
plt.figure(figsize=(12, 6))
plt.hist(s_power, bins=30, density=True, edgecolor='black', alpha=0.7, label='Histogram of Power')

# Plot Exponential PDF
x = np.linspace(min(s_power), max(s_power), 1000)
pdf = expon.pdf(x, scale=mean_s_power)
plt.plot(x, pdf, 'r', linewidth=2, label='Exponential PDF')

# Mark mean and +1 standard deviation on the plot
plt.axvline(mean_s_power, color='g', linestyle='--', linewidth=2, label=r'$\mu$ (Mean)')
plt.axvline(mean_s_power + std_dev_s_power, color='b', linestyle='--', linewidth=2, label=r'$\mu + \sigma$')

# Annotate mean and standard deviations outside the x-axis

offset = -0.00002 # Adjust this value to position the text outside the graph
plt.text(mean_s_power, offset, r'$\mu$', color='g', horizontalalignment='center')
plt.text(mean_s_power + std_dev_s_power, offset, r'$\mu + \sigma$', color='b', horizontalalignment='center')

# Labels and title
plt.xlabel('Power')
plt.ylabel(' Probability Density')
plt.title('Histogram of Power Values of SOUTH with Exponential PDF Overlay')
plt.legend()
plt.grid(True)

# Show plot
plt.show()
```



In [16]:
```
#STEP 15 = TALING FFT OF NORTH AND SOUTH DATA AND PLOTTING WITH FREQUENCY

# Parameters
Nf = 256       # Number of frequency bins(own choice)
NFFT = 2 * Nf  # Number of voltage samples to obtain FFT (512 in this case)
dt = 30.30e-9  # Time-resolution of voltage sampling in seconds (30.30 ns)
data_length = len(df)

# Determine the number of segments
num_segments = data_length // NFFT  # Number of segments(or bags ,where each bag can vcontain only 512 samples)
print(f"num_segment: {num_segments}")
# Initialize arrays to hold power spectra for N and S
power_spectra_N = np.zeros((num_segments, Nf))
power_spectra_S = np.zeros((num_segments, Nf))

# Compute the power spectrum for each segment
for i in range(num_segments):
    segment_N = df["N"].values[i * NFFT:(i + 1) * NFFT]
    segment_S = df["S"].values[i * NFFT:(i + 1) * NFFT]

    fft_N = np.fft.fft(segment_N, n=NFFT)
    fft_S = np.fft.fft(segment_S, n=NFFT)

    power_spectra_N[i, :] = np.abs(fft_N[:Nf]) ** 2
    power_spectra_S[i, :] = np.abs(fft_S[:Nf]) ** 2
   # power_spectra_CORELATED[i,:]= np.abs(fft_S[:Nf])
# Average the power spectra
average_power_spectrum_N = np.mean(power_spectra_N, axis=0)
average_power_spectrum_S = np.mean(power_spectra_S, axis=0)

# Compute the frequency bins
freqs = np.fft.fftfreq(NFFT, dt)[:Nf]
freqs = freqs+326.5e+6
# Reverse the power spectrum for Column S (flip vertically)
#average_power_spectrum_S = np.flip(average_power_spectrum_S)



# Plot the averaged power spectrum for N
plt.figure(figsize=(12, 6))
plt.plot(freqs, average_power_spectrum_N, label='Column N')
plt.title('Average Power Spectrum for Column N')
plt.xlabel('Frequency (Hz)')
plt.ylabel('Power')
plt.grid(True)
plt.legend()
plt.show()
# Plot the averaged (vertically flipped) power spectrum for S
plt.figure(figsize=(12, 6))
plt.plot(freqs, average_power_spectrum_S, label='Column S ', color='orange')
plt.title('Average Power Spectrum for Column S ')
plt.xlabel('Frequency (Hz)')
plt.ylabel('Power')
plt.grid(True)
plt.legend()
plt.show()



# Plot the averaged power spectrum for S
#plt.figure(figsize=(12, 6))
#plt.plot(freqs, average_power_spectrum_S, label='Column S', color='orange')
#plt.title('Average Power Spectrum for Column S')
#plt.xlabel('Frequency (Hz)')
#plt.ylabel('Power')
#plt.grid(True)
#plt.legend()
#plt.show()

# Output key values
print(f"Number of segments: {num_segments}")
print(f"Number of data points used per segment: {NFFT}")
print(f"Number of averaged spectra: {num_segments}")
```

```
num_segment: 60000
```





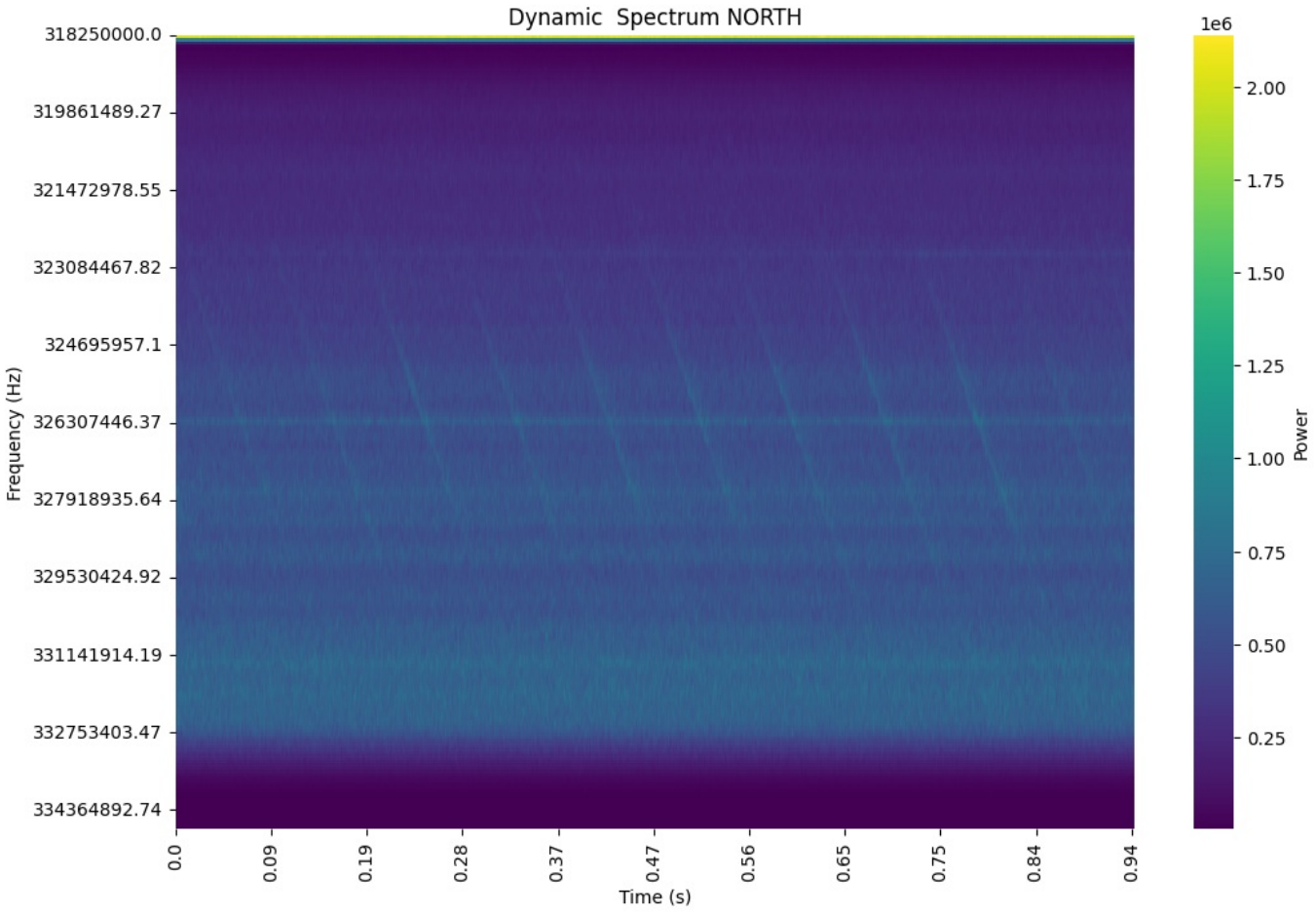

```
Number of segments: 60000
Number of data points used per segment: 512
Number of averaged spectra: 60000
```

In [17]:

```python
#STEP 16 = PLOTTING DYNAMIC SPECTRUM OF NORTH  DATA

#from scipy.ndimage import gaussian_filter

# Function to co-add spectra

def coadd_spectra(power_spectra, bin_size):
    num_segments = power_spectra.shape[0]
    num_bins = num_segments // bin_size
    print(f" num_bins:{num_bins}")
    coadded_spectra = np.zeros((num_bins, power_spectra.shape[1]))

    for i in range(num_bins):
        coadded_spectra[i, :] = np.mean(power_spectra[i*bin_size:(i+1)*bin_size, :], axis=0)
```

```
    return coadded_spectra

# Example parameters
NFFT = 512          # Number of FFT points
Nf = 256            # Number of frequency bins (Nf)
dt = 30.30e-9       # Time resolution of each spectral bin
dtN = 0.5e-3        # Decrease this to improve time resolution after co-adding

# Calculate bin size for co-adding
bin_size = int(dtN / (dt * NFFT))
print(f" binsize:{bin_size}")

# Ensure the bin size is valid
if bin_size < 1:
    bin_size = 1

# Co-add spectra
coadded_power_spectra_N = coadd_spectra(power_spectra_N, bin_size)

# Time axis for the plot
time_axis = np.arange(coadded_power_spectra_N.shape[0]) * dtN

# Frequency axis
freqs = np.fft.fftfreq(NFFT, dt)[:Nf]
freqs = freqs+318.25e+6
# Optional: Apply Gaussian smoothing to highlight features
coadded_power_spectra_N = gaussian_filter(coadded_power_spectra_N, sigma=1)

# Plot the dynamic spectrum
plt.figure(figsize=(12, 8))

# Define xticks and xticklabels safely
xtick_step = max(1, len(time_axis) // 10)
xticks = np.arange(0, len(time_axis), step=xtick_step)
xticklabels = np.round(time_axis[xticks], 2)

# Define yticks and yticklabels safely
ytick_step = max(1, len(freqs) // 10)
yticks = np.arange(0, len(freqs), step=ytick_step)
yticklabels = np.round(freqs[yticks], 2)

sns.heatmap(coadded_power_spectra_N.T, cmap='viridis', cbar_kws={'label': 'Power'},
            xticklabels=xticklabels, yticklabels=yticklabels)
plt.title('Dynamic  Spectrum NORTH')
plt.xlabel('Time (s)')
plt.ylabel('Frequency (Hz)')
plt.xticks(ticks=xticks, labels=xticklabels)
plt.yticks(ticks=yticks, labels=yticklabels)
#plt.gca().invert_yaxis()  # Invert y-axis to match the typical layout
plt.show()
```

```
binsize:32
num_bins:1875
```

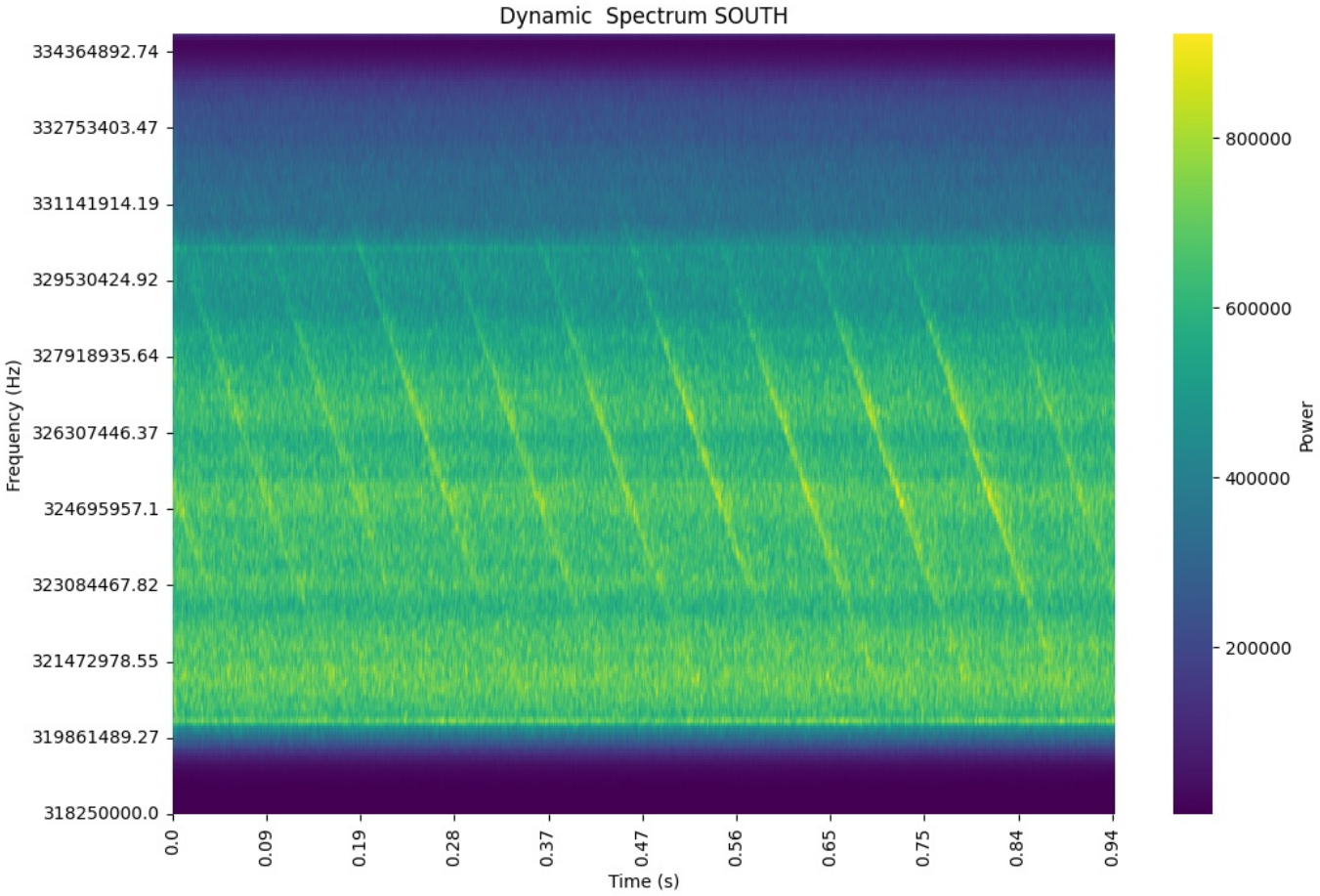


In [18]:

```
#STEP17  = PLOTTING DYNAMIC SPECTRUM OF SOUTH DATA

#from scipy.ndimage import gaussian_filter

# Function to co-add spectra

def coadd_spectra(power_spectra, bin_size):
    num_segments = power_spectra.shape[0]
    num_bins = num_segments // bin_size
    print(f" num_bins:{num_bins}")
    coadded_spectra = np.zeros((num_bins, power_spectra.shape[1]))

    for i in range(num_bins):
        coadded_spectra[i, :] = np.mean(power_spectra[i*bin_size:(i+1)*bin_size, :], axis=0)

    return coadded_spectra
```

```
# Example parameters
NFFT = 512          # Number of FFT points
Nf = 256            # Number of frequency bins (Nf)
dt = 30.30e-9       # Time resolution of each spectral bin
dtN = 0.5e-3        # Decrease this to improve time resolution after co-adding

# Calculate bin size for co-adding
bin_size = int(dtN / (dt * NFFT))
print(f" binsize:{bin_size}")

# Ensure the bin size is valid
if bin_size < 1:
    bin_size = 1
#adding north south
power_spectra_T=(power_spectra_N)+(power_spectra_S)

# Co-add spectra
coadded_power_spectra_S = coadd_spectra(power_spectra_S, bin_size)

# Co-add spectra
coadded_power_spectra_T = coadd_spectra(power_spectra_T, bin_size)



# Time axis for the plot
time_axis = np.arange(coadded_power_spectra_S.shape[0]) * dtN

# Frequency axis
freqs = np.fft.fftfreq(NFFT, dt)[:Nf]
freqs = freqs+318.25e+6

# Optional: Apply Gaussian smoothing to highlight features
coadded_power_spectra_S = gaussian_filter(coadded_power_spectra_S, sigma=1)

# Optional: Apply Gaussian smoothing to highlight features
coadded_power_spectra_T = gaussian_filter(coadded_power_spectra_T, sigma=1)

# Plot the dynamic spectrum
plt.figure(figsize=(12, 8))

# Define xticks and xticklabels safely
xtick_step = max(1, len(time_axis) // 10)
xticks = np.arange(0, len(time_axis), step=xtick_step)
xticklabels = np.round(time_axis[xticks], 2)

# Define yticks and yticklabels safely
ytick_step = max(1, len(freqs) // 10)
yticks = np.arange(0, len(freqs), step=ytick_step)
yticklabels = np.round(freqs[yticks], 2)
coadded_power_spectra_S=np.flip(coadded_power_spectra_S,axis=1)
sns.heatmap(coadded_power_spectra_S.T, cmap='viridis', cbar_kws={'label': 'Power'},
            xticklabels=xticklabels, yticklabels=yticklabels)
plt.title('Dynamic  Spectrum SOUTH')
plt.xlabel('Time (s)')
plt.ylabel('Frequency (Hz)')
plt.xticks(ticks=xticks, labels=xticklabels)
plt.yticks(ticks=yticks, labels=yticklabels)
plt.gca().invert_yaxis()  # Invert y-axis to match the typical layout
plt.show()
```

```
binsize:32
num_bins:1875
num_bins:1875
```

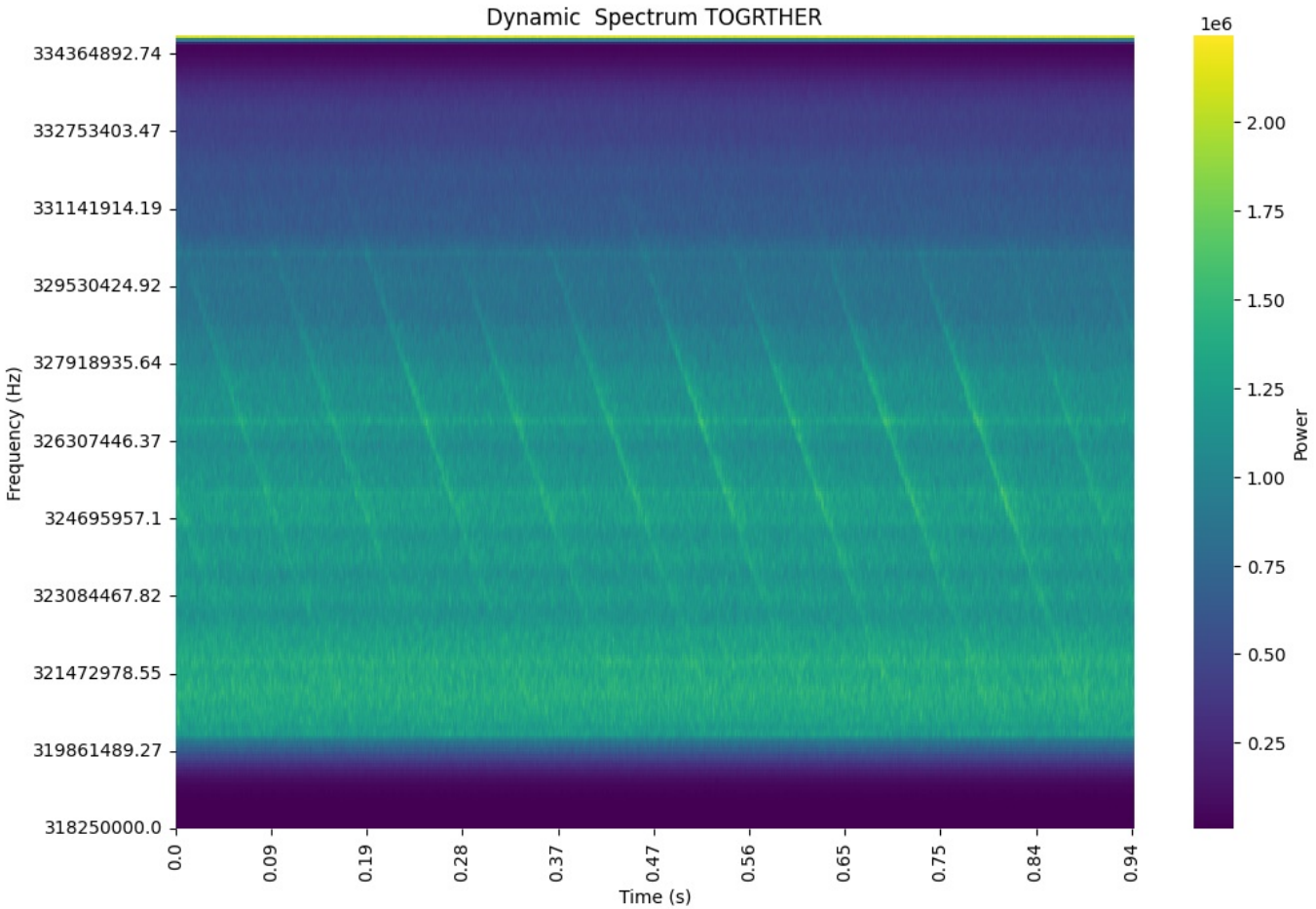


In [19]:

```python
#STEP18  = PLOTTING DYNAMIC SPECTRUM OF COADDED DATA(NORTH POWER+SOUTH POWER)

#from scipy.ndimage import gaussian_filter

# Function to co-add spectra

def coadd_spectra(power_spectra, bin_size):
    num_segments = power_spectra.shape[0]
    num_bins = num_segments // bin_size
    print(f" num_bins:{num_bins}")
    coadded_spectra = np.zeros((num_bins, power_spectra.shape[1]))

    for i in range(num_bins):
        coadded_spectra[i, :] = np.mean(power_spectra[i*bin_size:(i+1)*bin_size, :], axis=0)

    return coadded_spectra

# Example parameters
NFFT = 512          # Number of FFT points
Nf = 256            # Number of frequency bins (Nf)
dt = 30.30e-9       # Time resolution of each spectral bin
dtN = 0.5e-3        # Decrease this to improve time resolution after co-adding

# Calculate bin size for co-adding
bin_size = int(dtN / (dt * NFFT))
print(f" binsize:{bin_size}")
```

```
# Ensure the bin size is valid
if bin_size < 1:
    bin_size = 1
#adding north south
power_spectra_T=(power_spectra_N)+(power_spectra_S)

# Co-add spectra
#coadded_power_spectra_S = coadd_spectra(power_spectra_S, bin_size)

# Co-add spectra
coadded_power_spectra_T = coadd_spectra(power_spectra_T, bin_size)



# Time axis for the plot
time_axis = np.arange(coadded_power_spectra_T.shape[0]) * dtN

# Frequency axis
freqs = np.fft.fftfreq(NFFT, dt)[:Nf]
freqs = freqs+318.25e+6

# Optional: Apply Gaussian smoothing to highlight features
#coadded_power_spectra_S = gaussian_filter(coadded_power_spectra_S, sigma=1)

# Optional: Apply Gaussian smoothing to highlight features
coadded_power_spectra_T = gaussian_filter(coadded_power_spectra_T, sigma=1)

# Plot the dynamic spectrum
plt.figure(figsize=(12, 8))

# Define xticks and xticklabels safely
xtick_step = max(1, len(time_axis) // 10)
xticks = np.arange(0, len(time_axis), step=xtick_step)
xticklabels = np.round(time_axis[xticks], 2)

# Define yticks and yticklabels safely
ytick_step = max(1, len(freqs) // 10)
yticks = np.arange(0, len(freqs), step=ytick_step)
yticklabels = np.round(freqs[yticks], 2)
coadded_power_spectra_T=np.flip(coadded_power_spectra_T,axis=1)
sns.heatmap(coadded_power_spectra_T.T, cmap='viridis', cbar_kws={'label': 'Power'},
            xticklabels=xticklabels, yticklabels=yticklabels)
plt.title('Dynamic  Spectrum TOGRTHER')
plt.xlabel('Time (s)')
plt.ylabel('Frequency (Hz)')
plt.xticks(ticks=xticks, labels=xticklabels)
plt.yticks(ticks=yticks, labels=yticklabels)
plt.gca().invert_yaxis()  # Invert y-axis to match the typical layout
plt.show()
```

```
binsize:32
num_bins:1875
```



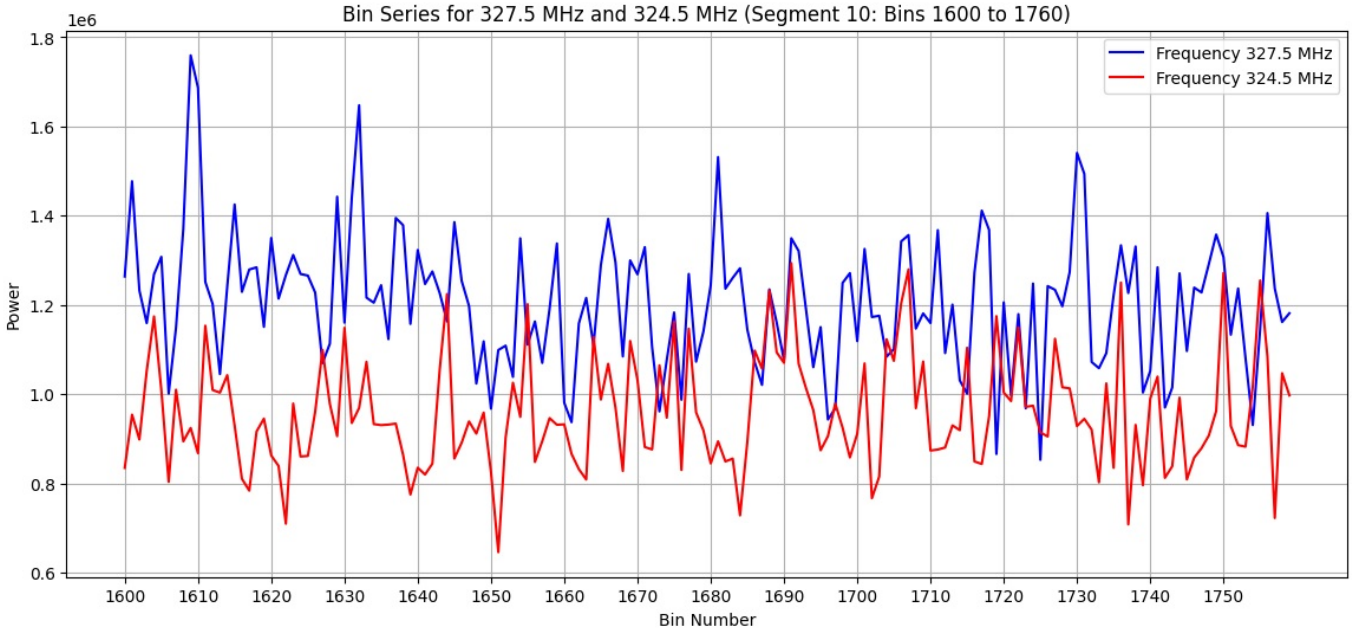
In [20]:

```python
#STEP19  = FINDING DISPERSION MEASURE FROM 3 DIFF TIME SERIES PLOT ACROSS 2 FREQ BINS(COADDED)

# Function to co-add spectra
def coadd_spectra(power_spectra, bin_size):
    num_segments = power_spectra.shape[0]
    num_bins = num_segments // bin_size
    coadded_spectra = np.zeros((num_bins, power_spectra.shape[1]))

    for i in range(num_bins):
        coadded_spectra[i, :] = np.mean(power_spectra[i*bin_size:(i+1)*bin_size, :], axis=0)

    return coadded_spectra

# Example parameters
NFFT = 512          # Number of FFT points
```

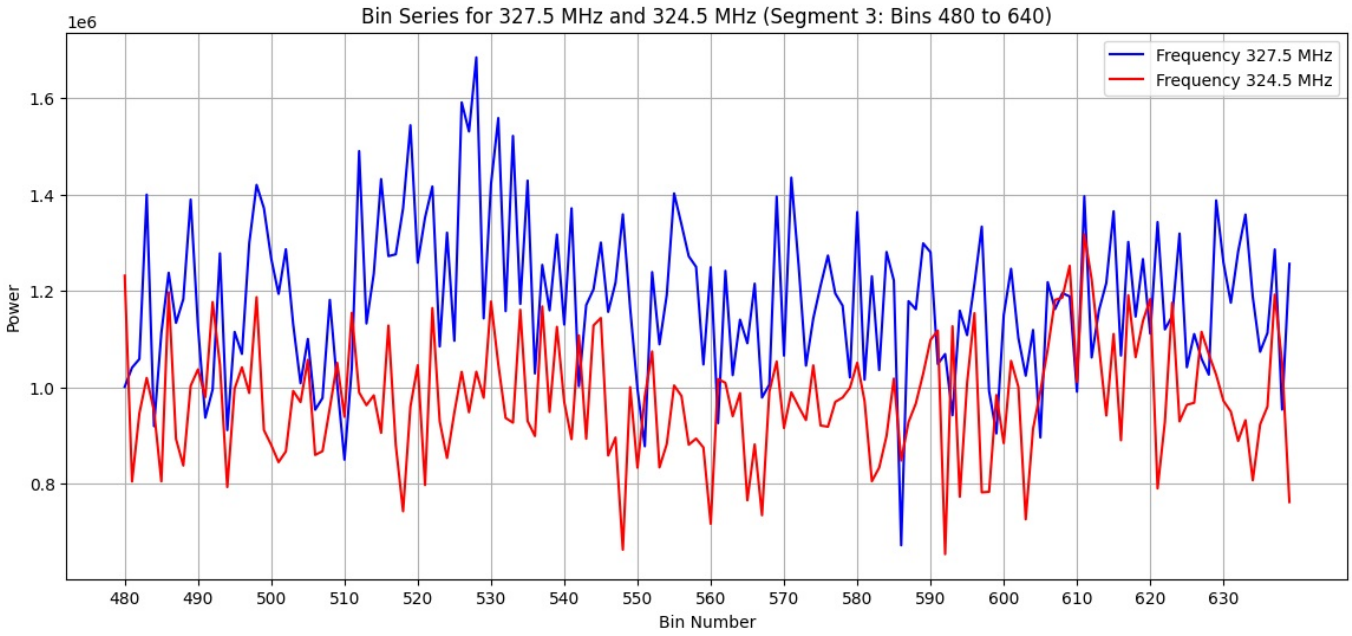
```
Nf = 256            # Number of frequency bins (Nf)
dt = 30.30e-9       # Time resolution of each spectral bin
dtN = 0.5e-3        # Decrease this to improve time resolution after co-adding

# Calculate bin size for co-adding
bin_size = int(dtN / (dt * NFFT))

# Ensure the bin size is valid
if bin_size < 1:
    bin_size = 1

# Assuming power_spectra_S is your input data matrix
# Co-add spectra
coadded_power_spectra_T = coadd_spectra(power_spectra_T, bin_size)

# Bin axis (number of bins)
bin_axis = np.arange(coadded_power_spectra_T.shape[0])

# Frequency axis (in Hz)
freqs = np.fft.fftfreq(NFFT, dt)[:Nf] + 318.25e6

# Frequencies in MHz
freq1 = 327.5e6
freq2 = 324.5e6

# Find the index for each frequency
index1 = np.argmin(np.abs(freqs - freq1))
index2 = np.argmin(np.abs(freqs - freq2))

# Define the size of the segments
segment_size = 160
num_segments = coadded_power_spectra_T.shape[0] // segment_size

# Specify the segments to plot
segments_to_plot = [10, 3, 4]

# Calculate frequency difference squared in MHz^2
delta_freq = (freq1/1e6)**2 - (freq2/1e6)**2
delta_freq *= 4.149e3

# Initialize a list to store the DM values
dm_values = []

# Loop through each specified segment and analyze
for i, segment in enumerate(segments_to_plot):
    start_bin = segment * segment_size
    end_bin = start_bin + segment_size

    # Extract the bin series for the given frequencies within the current segment
    bin_series_freq1 = coadded_power_spectra_T[start_bin:end_bin, index1]
    bin_series_freq2 = coadded_power_spectra_T[start_bin:end_bin, index2]

    # Find the index of the maximum peak for each frequency
    max_index1 = np.argmax(bin_series_freq1)

    # Special handling for segment 11
    if segment == 11:
        # Look for a red peak that is to the right of the blue peak
        potential_peaks = np.where(bin_series_freq2[max_index1+1:] == np.max(bin_series_freq2[max_index1+1:]))[0
        if len(potential_peaks) > 0:
            max_index2 = max_index1 + 1 + potential_peaks[0]
        else:
            max_index2 = np.argmax(bin_series_freq2)  # Fallback if no peak is found to the right
    else:
        max_index2 = np.argmax(bin_series_freq2)

    # Calculate the bin difference
    bin_difference = (start_bin + max_index2) - (start_bin + max_index1)

    # Calculate the time delay
    time_delay = 0.0005 * bin_difference

    # Calculate DM
    prod = (freq1/1e6) * (freq2/1e6)
    prod **= 2
    DM = (time_delay * prod) / delta_freq

    # Store the DM value
    dm_values.append(DM)

    # Print the results for the current segment
    print(f"Segment {segment}:")
    print(f"  Maximum peak for {freq1/1e6} MHz at bin: {start_bin + max_index1}")
```

```
    print(f"  Maximum peak for {freq2/1e6} MHz at bin: {start_bin + max_index2}")
    print(f"  Difference in bin numbers between peaks: {bin_difference}")
    print(f"  Time delay: {time_delay} seconds")
    print(f"  Dispersion Measure (DM): {DM:.2f} pc cm^-3")

    # Plot bin series for both frequencies in the current segment
    plt.figure(figsize=(14, 6))
    plt.plot(bin_axis[start_bin:end_bin], bin_series_freq1, label=f'Frequency {freq1/1e6} MHz', color='blue')
    plt.plot(bin_axis[start_bin:end_bin], bin_series_freq2, label=f'Frequency {freq2/1e6} MHz', color='red')
    plt.xticks(np.arange(start_bin, end_bin, 10))
    plt.xlabel('Bin Number')
    plt.ylabel('Power')
    plt.title(f'Bin Series for 327.5 MHz and 324.5 MHz (Segment {segment}: Bins {start_bin} to {end_bin})')
    plt.legend()
    plt.grid(True)

    # If this is the last segment in the list, calculate and display the average DM
    if i == len(segments_to_plot) - 1:
        average_dm = np.mean(dm_values)
        plt.figtext(0.5, 0.01, f'Average DM: {average_dm:.2f} pc cm^-3', ha="center", fontsize=12, bbox={"facec

    plt.show()
```

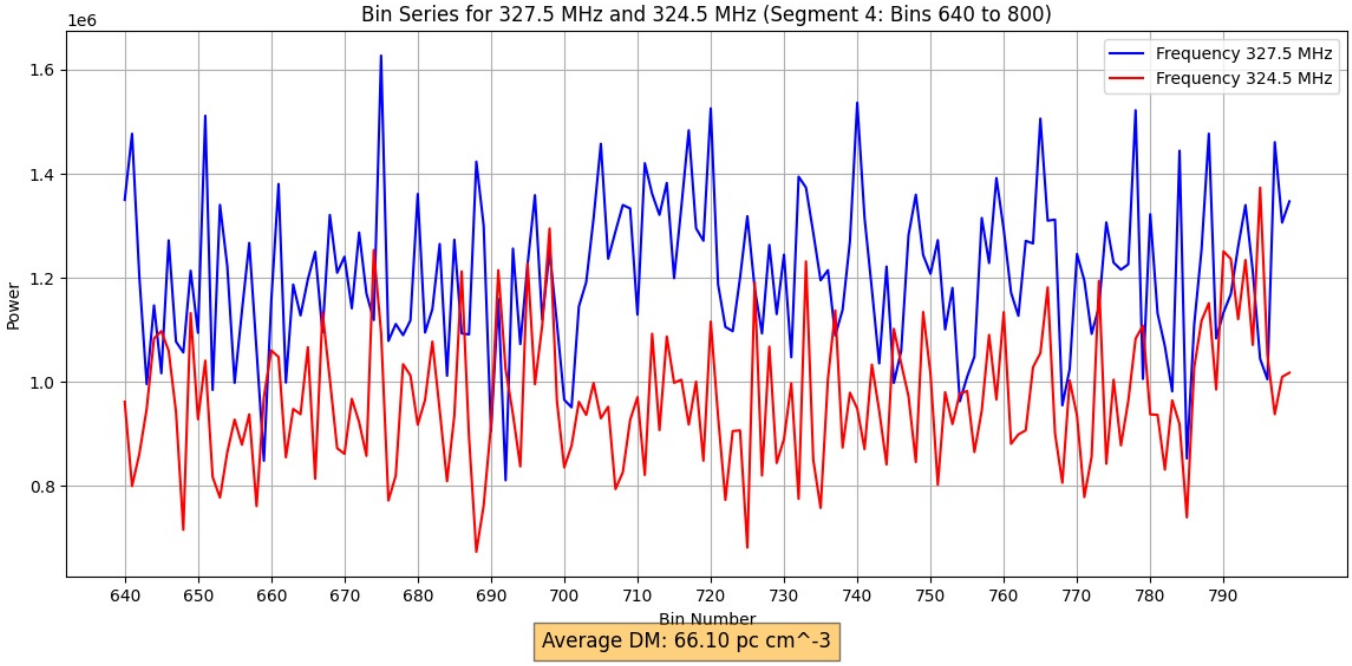
```
Segment 10:
  Maximum peak for 327.5 MHz at bin: 1609
  Maximum peak for 324.5 MHz at bin: 1691
  Difference in bin numbers between peaks: 82
  Time delay: 0.041 seconds
  Dispersion Measure (DM): 57.06 pc cm^-3
```




```
Segment 3:
  Maximum peak for 327.5 MHz at bin: 528
  Maximum peak for 324.5 MHz at bin: 611
  Difference in bin numbers between peaks: 83
  Time delay: 0.0415 seconds
  Dispersion Measure (DM): 57.75 pc cm^-3
```

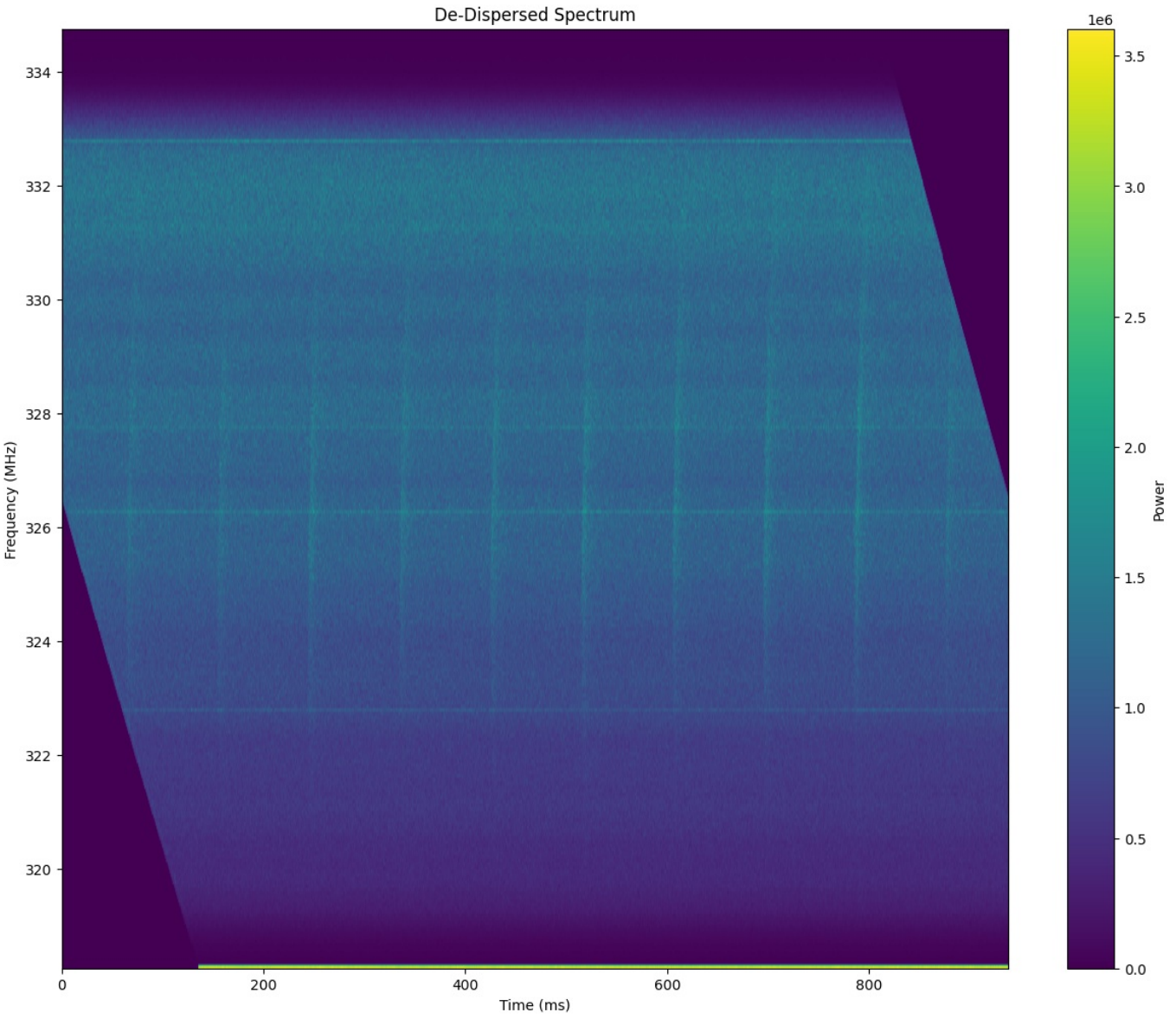



```
Segment 4:
  Maximum peak for 327.5 MHz at bin: 675
  Maximum peak for 324.5 MHz at bin: 795
  Difference in bin numbers between peaks: 120
  Time delay: 0.06 seconds
  Dispersion Measure (DM): 83.50 pc cm^-3
```



In [21]:
```
#STEP20  = PLOTTING DE-DISPERSED DYNAMIC SPECTRUM (COADDED)
```

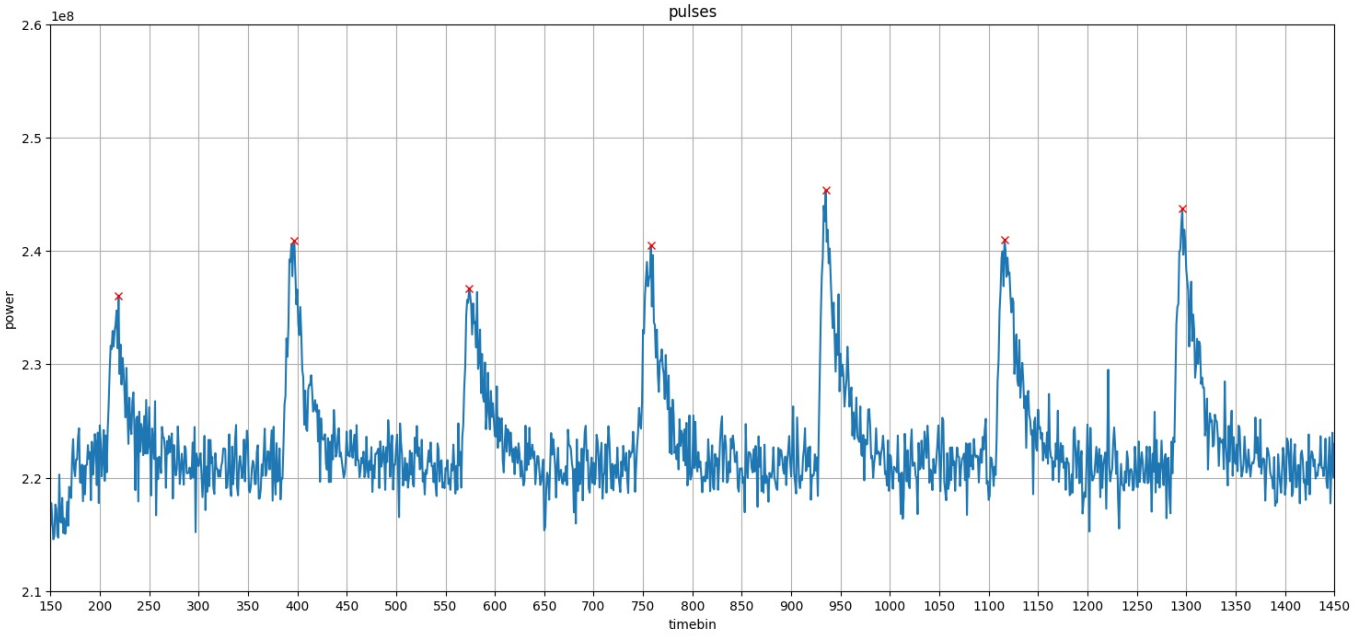
```
#from scipy.ndimage import shift

# Parameters
DM = 66.10 # Dispersion Measure in pc cm^-3
central_freq = 326.5  # Central reference frequency in MHz
bandwidth = 16.50  # Bandwidth in MHz
freq_bins = 256  # Number of frequency bins
time_bins = coadded_power_spectra_T.shape[0]  # Number of time bins

# Frequency axis (in MHz)
freq = np.linspace(central_freq - bandwidth/2, central_freq + bandwidth/2, freq_bins)

# De-dispersed spectrum initialization
deDispersed = np.zeros((freq_bins, time_bins))

# Transpose the coadded power spectrum for easy indexing
cross_spectra = np.transpose(coadded_power_spectra_T)
# Calculate the delay for each frequency bin relative to the central frequency
for i in range(freq_bins):
    delay = 4.149e3 * DM * ((freq[i]**(-2)) - (central_freq**(-2)))
    deDispersed[i, :] = shift(cross_spectra[i, :], shift=int(delay / (dtN )))

# Plot the de-dispersed spectrum
plt.figure(figsize=(15, 12))
plt.imshow(deDispersed, aspect='auto', origin='lower', cmap='viridis',
           extent=[0, time_bins * dtN * 1e3, freq[0], freq[-1]])
plt.colorbar(label='Power')
plt.title("De-Dispersed Spectrum")
plt.xlabel("Time (ms)")
plt.ylabel("Frequency (MHz)")
plt.show()
```



In [22]:
```
## #STEP 21 = PLOTTING DE-DISPERSED TIME SERIES OF POWER (COADDED),CALCULATING TIME PERIOD
```

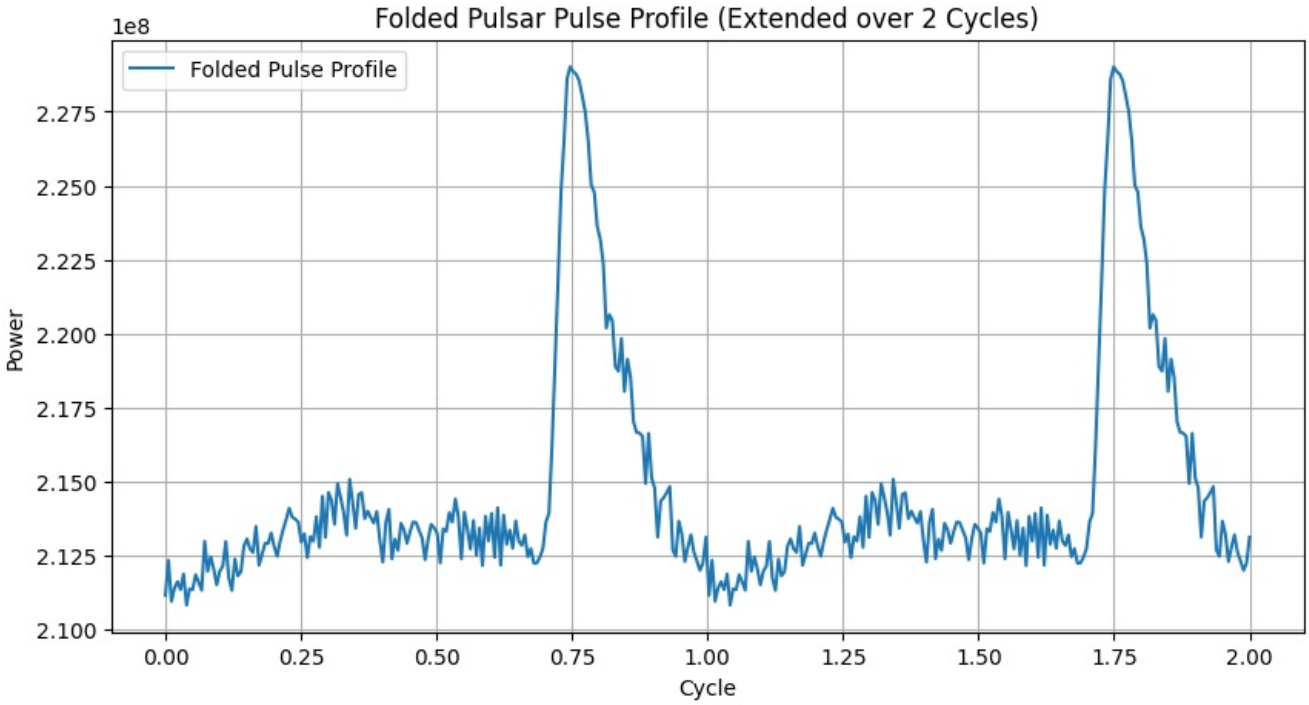
```python
# Assuming 'deDispersed' is your data array
start = 100
end = 1600
pulses = deDispersed.sum(0)[start:end]

# Plotting the time series
plt.figure(figsize=(18, 8))
plt.plot(pulses)
plt.xticks(np.arange(0, end-start, 50))  # Adjust the xticks to match the sliced data
plt.grid()

# Finding peaks with a height filter and minimum distance
peaks, properties = find_peaks(pulses, height=2.3e+8, distance=100)  # Adjust 'distance' as needed

# Calculate the prominence of the found peaks
prominences = peak_prominences(pulses, peaks)[0]

# Select the top 8 peaks based on their prominence
prominent_peaks = np.argsort(prominences)[::-1][:8]
peak_indices = peaks[prominent_peaks]

# Sort the selected peaks by their position on the time axis
sorted_peak_indices = np.sort(peak_indices)

# Get the corresponding time positions of the sorted peaks
peak_times = sorted_peak_indices + start

# Print the time positions of the 8 most prominent peaks (sorted by time)
print("Timebin positions of the 8 most prominent peaks:")
for i, time in enumerate(peak_times, start=1):
    print(f"Peak {i} = {time}")

# Calculate the differences between adjacent peaks
peak_diffs = np.diff(peak_times)

# Print the time differences between adjacent peaks
print("\nTimebin differences between adjacent major peaks:")
for i, diff in enumerate(peak_diffs, start=1):
    print(f"Difference between Peak {i+1} and Peak {i} = {diff} timebin units")

# Calculate and print the average difference
average_diff = np.mean(peak_diffs)
print("\nAverage difference between major peaks:", average_diff,"timebin")
#timeperiod
time_period=average_diff*0.5         #my time resolution 0.5milisecond
print("time period :",time_period ,"mili second")
# Plotting the prominent peaks on the time series
plt.plot(sorted_peak_indices, pulses[sorted_peak_indices], "x", color="red")
plt.ylim(2.1e8,2.6e8)
plt.xlim(150,1450)
plt.xlabel("timebin")
plt.ylabel("power")
plt.title("pulses")
plt.show()
```

```
Timebin positions of the 8 most prominent peaks:
Peak 1 = 319
Peak 2 = 497
Peak 3 = 674
Peak 4 = 858
Peak 5 = 1035
Peak 6 = 1216
Peak 7 = 1396
Peak 8 = 1575

Timebin differences between adjacent major peaks:
Difference between Peak 2 and Peak 1 = 178 timebin units
Difference between Peak 3 and Peak 2 = 177 timebin units
Difference between Peak 4 and Peak 3 = 184 timebin units
Difference between Peak 5 and Peak 4 = 177 timebin units
Difference between Peak 6 and Peak 5 = 181 timebin units
Difference between Peak 7 and Peak 6 = 180 timebin units
Difference between Peak 8 and Peak 7 = 179 timebin units

Average difference between major peaks: 179.42857142857142 timebin
time period : 89.71428571428571 mili second
```



In [23]:

```
#STEP 22 = PLOTTING FOLDED  POWER PROFILE FOR 2 CYCLE

# Assume `pulses` is defined as a subset of `deDispersed.sum(0)[start:end]`
start = 0
end = 2000
pulses = deDispersed.sum(0)[start:end]

# Define parameters
time_period = 89.7 / 1000  # 89.7 milliseconds converted to seconds
total_duration = 1  # Total duration is 1 second
total_bins = 180
phase_bins = total_bins

# Calculate the number of cycles that fit into the pulses data
num_cycles = len(pulses) // phase_bins

# Trim pulses to make it divisible by the number of phase_bins
pulses_trimmed = pulses[:num_cycles * phase_bins]

# Reshape the trimmed data into a 2D array
data_2d = pulses_trimmed.reshape((num_cycles, phase_bins))

# Average power row-wise
averaged_power = np.mean(data_2d, axis=0)

# Plotting the averaged power vs phase bins for one cycle
plt.figure(figsize=(10, 5))

# Plot for two cycles
x_values = np.linspace(0, 2, 2 * phase_bins)  # Normalized to two cycles
y_values = np.tile(averaged_power, 2)  # Repeat the averaged power values

plt.plot(x_values, y_values, label="Folded Pulse Profile")

# Labeling the plot
plt.xlabel('Cycle')
plt.ylabel('Power')
plt.title('Folded Pulsar Pulse Profile (Extended over 2 Cycles)')
plt.grid(True)
plt.legend()
plt.show()
```



In [ ]:

In [ ]: